

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**MULTILINGUAL USER INTERFACE FOR AN  
OPERATING SYSTEM**

Inventor(s):

**Edward S. Miller**  
**Bjorn C. Rettig**  
**Gregory Wilson**  
**Shan Xu**  
**Arul A. Menezes**  
**Michael J. Thomson**  
**Sharad Mathur**  
**Roberto Cazzaro**  
**Michael Ginsberg**

ATTORNEY'S DOCKET NO. MS1-332USC1

**EV 395542210**

1      **RELATED APPLICATIONS**

2      This application is a continuation of United States Patent Application Serial  
3      Number 09/596,236, filed 6/13/2000, entitled "Multilingual User Interface for an  
4      Operating System", by inventors Edward S. Miller, Bjorn C. Rettig, Gregory  
5      Wilson, Shan Xu, Arul A. Menezes, Michael J. Thomson, Sharad Mathur, Roberto  
6      Cazzaro, and Michael Ginsberg, which is a continuation-in-part of a United States  
7      Patent Application Serial Number 09/134,559, filed 8/14/98, entitled "Multilingual  
8      User Interface for an Operating System", by inventors Edward S. Miller, Bjorn C.  
9      Rettig, Gregory Wilson, and Shan Xu, both of which applications are incorporated  
10     by reference.

11     **TECHNICAL FIELD**

12     The present invention generally relates to operating systems and more  
13     particularly to operating systems that provide an efficient mechanism for  
14     switching the user-interface language.

15     **BACKGROUND OF THE INVENTION**

16     A resource is binary data or non-binary data, e.g., a text file. In Windows  
17     NT® and all other O/S of the Windows® family, resources are binary data.  
18     Resource data can reside in the executable file of an application, so the executable  
19     file is a binary file with code and resource data in it. Processes defined by the  
20     code can use the resources in their own binary executable files or other executable  
21     files. Resources used by such processes may also reside in resource-only files, for  
22     example, resource-only dynamic link libraries (DLLs). A resource may be either  
23     standard or user-defined. The data in a standard resource describes an icon,  
24  
25

1 cursor, menu, dialog box, bitmap, enhanced metafile, font, accelerator table,  
2 message-table entry, string-table entry, or version. A user-defined resource  
3 contains any data required by a specific application. The resources required by  
4 operating system processes may be handled in various different ways. Many of  
5 these resources include words, symbols, formatting data, etc. that are language-  
6 specific. Usually, a particular language is determined by the operating system  
7 installation package chosen by the user. If the language of the software is English,  
8 only the English-language-specific resources will be installed with the operating  
9 system. This is convenient because of the large quantity of language-specific  
10 resources that would have to be copied on the hard-disk or other non-volatile  
11 memory to cover all languages.

12 Providing a single language for the operating system to support is also  
13 convenient because it allows resources to be efficiently loaded and unloaded into  
14 and from memory as the need arises. Far too many resources exist for all to reside  
15 in memory at all times. To manage the loading and unloading of resources so that  
16 resources do not unnecessarily occupy memory when not required, the code that  
17 generates the processes requiring the resources and the resources peculiar to the  
18 process may be incorporated in the same binary files. When a process is invoked,  
19 a binary file containing the code for the process, and the attendant resources, may  
20 be loaded into memory or otherwise made accessible to the process. When the  
21 process is terminated, the resource and code sections of such a file are unloaded  
22 from memory or otherwise made inaccessible. These binary files can be  
23 executable programs, dynamic link libraries (DLLs), device drivers, etc. If they  
24 were bloated with all the alternative language resources, an excessive amount of  
25 memory would be required.

1 An example of how one operating system handles such resources is as  
2 follows. First, a resource finder, an operating system function, is employed to  
3 create a handle to the specified resource's info block. A process requiring a  
4 resource sends the finder a resource module handle and the resource name, type,  
5 and optionally, a language ID. The latter specifies a language specific resource in  
6 the resources defined by the resource module handle. The finder returns a handle  
7 to the specified resource's info block and the process can call a resource loader to  
8 place the resource in memory. The process gives the resource handle and the  
9 resource module handle to the resource loader, which places the resource in  
10 memory and returns a handle to the memory block containing the resource. The  
11 resource is then available to the process. The operating system may then use other  
12 devices to free the memory after the process loading it into memory no longer  
13 needs it, is terminated, or if other conditions require it.

14 The above is only one type of resource access facility in an example  
15 operating system. Other mechanisms may make resources available in other ways,  
16 such as by placing text messages in an output buffer, immediately loading and  
17 returning a handle to resource data in a single function call, etc. The common  
18 feature of these mechanisms is that they find a resource either in memory or in a  
19 disk file or other storage system and make the resource available to the process  
20 that requires it. This may involve loading a file from disk into memory or just  
21 providing access to the resource by providing a handle or some other device. The  
22 file (, device, or channel) containing the resource may be in the same file as the  
23 code defining the requesting process or another file. The other file could contain  
24 code or be a resource-only file. A process may not need explicitly to unload a  
25 resource it no longer needs.

With the low cost of disk storage, it may be desirable in some instances for the same installation of an operating system to provide, transparently to the user, appropriate resources for a number of languages. However, for an operating system built around the above resource management regimes, the options available to modify the operating system to accommodate selectable languages appear quite problematic, as discussed below.

To provide multilingual support, one option might be to provide a different set of binary files for each language. Considering there might be on the order of a thousand binary files containing language-specific resources in an complex operating system and that it might be desired to support many different languages, the number of binary files to be installed would be large indeed. In addition to the labor required to provide for the selection of a language by the user, the redundancy in the resulting mass of files would be tremendous because all language-non-specific resources would be duplicated for each language supported. Not only would the language-non-specific resource require duplication, but also all the code sections.

Another option might be to install the operating system binary files anew, each time a new user requiring a different language logged on. This option is unattractive because it would take a great deal of time.

Still another option might be to provide the different language-specific resources in each binary file. This would eliminate the redundancy of the first option since each binary file would only add language-specific resources. However, this option would require recoding of each binary file, so it also is not an elegant option. Something similar to this is currently done on a very limited basis. Some binary files contain alternate resources, each being preferred

1 depending on the language or country of the user. The code sections of these  
2 binary files define processes that address a different resource based on a “guess”  
3 as to the preferred language of resource. This guess is made based on the settings  
4 of some system parameter, for example, which date format has been selected. So,  
5 for example, if a Russian style of date is selected, the resources tagged as Russian  
6 might be loaded.

7 There is at least one type of operating system that now provides for  
8 language selection on a limited basis. This operating system provides separate  
9 text files for each language. When a process requires a text file resource in a  
10 particular language, the operating system addresses the appropriate file. The user  
11 can select his default language of choice through a system variable.

12 As mentioned briefly, at least one current operating system (Windows®)  
13 provides some support for the creation of language-specific libraries, for example  
14 text messages. A system variable is defined indicating the locale (Note, the locale  
15 of a system is not a language setting. Locale is a mixture of language and  
16 location) of the operating system installation and this variable can be used by the  
17 applications running on the operating system to format messages specifically for  
18 the current language. This requires, however, that the process (the application)  
19 identify precisely the appropriate language resource and where it is located. As a  
20 model for conversion it would entail extensive recoding.

21 None of the prior art operating system regimes offers a model suggesting  
22 how to provide multilingual support by the operating system in a very automatic  
23 way. Also, none suggests means of preserving some of the inherent economies of  
24 binary files with code and resource sections in the same file. The simple  
25 transformations suggested above to provide the desired functionality appear to be

1 unduly expensive and/or bulky in terms of the redundant data required. Any  
2 conversion that is readily implemented would likely have to be a system that  
3 departs significantly from any of the prior art systems.

4 Referring to Fig. 1, in a common operation in a prior art operating system,  
5 a binary file 20 is loaded. The binary file 20 contains a code section 10 and a  
6 resource section 30 and may be any file unit of the operating system or one  
7 supplied by a third party. For example, the binary file 20 could be an executable  
8 binary, a dynamic link library (DLL), or a device driver. The resource section 30  
9 may contain some of the resources used by the code section, particularly those  
10 resources peculiar to the requirements of the processes generated by the code  
11 section 10 and which may be unloaded from memory when the processes defined  
12 in the code section 10 are no longer required. In other words, the resources 30 are  
13 those that may be required by processes encoded in the code section 10 and once  
14 those processes are terminated, there is no longer any need to maintain the  
15 resources contained in resource section 30 in memory. For example, the binary  
16 file 10 could be a core resource or an application that is supplied with the  
17 operating system such as a stripped-down text editor. For the editor, for example,  
18 when the user terminates the editor program, the resources required by this text  
19 editor would no longer be required. The binary file 20, including code 10 and  
20 resources 30, would be removed from memory. Of course, the code section 10  
21 could use other resources from other files and may also use other processes as  
22 well.

23 Referring to Fig. 2, resources 85 and the code 55 that uses it may also be  
24 located in separate respective files 25 and 22.. For example, the resource 85  
25 addressed by an application 55 defined in a piece of code 50 may be contained in a

1 resource-only DLL or a separate file 25 that contains code 70 and resources 60.  
2 The application 55 may reside in a file that also contains resources 40. Another  
3 operating system device may be used to find the file by resource type and name.  
4 The management (loading and unloading) of the resources may be handled by the  
5 resource loader.

6 Referring to Fig. 3, resources are addressed by a process 110 using a  
7 resource loader 130 and a resource finder 135. The resource loader is an operating  
8 system facility that provides access to a resource datum 125 given a resource  
9 module handle and resource handle. The resource module handle, which indicates  
10 where the resource datum, specified by the resource name, can be found, is created  
11 by the resource finder. The resource name, type, and a language (the latter is  
12 optional) are provided to the resource finder 135 which returns a resource module  
13 handle. If the resource is in a module other than the one that generated the calling  
14 process, the handle of that module must be provided to the resource finder as well.  
15 The resource type may specify for example a bitmap, an animated cursor, a font  
16 resource, a menu resource, a string-table entry, etc.

17 The resource loader loads the specified resource into memory. It requires a  
18 resource module handle and resource handle. The resource module handle is the  
19 module whose executable file contains the resource. The resource handle  
20 identifies the resource to be loaded. The resource loader 130 returns a handle to  
21 the memory block containing the data associated with the resource. The  
22 description shown in Fig. 3 is consistent with either of the situations shown in  
23 Fig. 1 or Fig. 2. Note that examples of the above functions are defined in  
24 documentation relating to the Windows® APIs FindResource and LoadResource.  
25 Note also that the resource may be loaded in a prior operation as well as part of a

1 call for a resource as described above. For example in the Windows® operating  
2 system, a call to LoadLibrary could result in the loading of a module into memory.

3 Referring to Fig. 4, a generalized schematic of how resources may be  
4 addressed in an operating system is shown. A resource handler 230 is used by a  
5 process 210 to obtain access to a resource datum 220. The resource handler 230  
6 may consist of several different devices provided by the operating system, for  
7 example as discussed with reference to Fig. 3. The process identifies the  
8 requested resource to the handler 230 and may tell the handler where the resource  
9 can be found, such as a file name, identifier of a module 250 or some other  
10 information. The resource handler 230 may need to load the resource 220,  
11 possibly included in a module 250, into memory or some other means for making  
12 the data accessible 240 providing access to the process 210. The process 210 is  
13 given a handle, address, pointer, etc. to access the resource 220. The important  
14 features of the process described by Fig. 4 are that the process identifies the  
15 resource required and the operating system provides the process with access to that  
16 resource. The resource may reside on a disk, on another computer connected by a  
17 network, provided through a communications port or any other mechanism for  
18 transferring data to a process on a computer. The operating system may, as part of  
19 the request, transfer the resource to a different medium, say, for example, from  
20 disk to memory, before access to the resource by the process is possible.

21

22 **SUMMARY OF THE INVENTION**

23 An operating system scheme provides resource-handling components that  
24 provide features for handling multiple-language resources without requiring any  
25 specific directions from the processes requesting the resources. This allows the

1 operating system to provide multilingual support while using existing resource and  
2 executable binary files without modification of these elements. That is, a user is  
3 enabled to select a language for the user interface and the resource loader will  
4 automatically redirect calls for resources to the appropriate resources.

5 Note that throughout the following description, the notion of loading data  
6 into memory is not intended to be construed literally as actually taking data from a  
7 file and putting it into memory. In the operating system context contemplated by  
8 the invention, the actual loading of data into physical memory is performed by low  
9 level operating system functions. Each process may have a virtual memory space  
10 that does not coincide with actual physical memory. When, in the following  
11 discussion, the step of loading and unloading data from memory is spoken of, it is  
12 intended to be interpreted broadly as any operating system function that makes  
13 data accessible to the process.

14 From the standpoint of the processes requesting resources, the interactions  
15 with operating system devices are the same as for handling resources of a single  
16 language. The operating system resource-handling components for finding  
17 resources and returning them to a requesting process are modified to dynamically  
18 generate a path to an alternate-language resource module. The generation of the  
19 path may be in response to a resource identifier and an optional module handle  
20 provided by the process requesting the resource and also in response to a system-  
21 wide operating user-setting specifying a chosen language for the user-interface.  
22 The path to the alternate-language resource is used instead of the module handle, if  
23 any, supplied by the process.

24 By generating the module handle dynamically, the operating system may be  
25 expanded without modifications to any permanent facility to correlate base module

handles (the ones used by the calling process) and the alternate-language resource modules. Since the look-up table is generated dynamically, it is automatically created for the purpose of saving steps and is never out of date. When new modules are added to the operating system, alternate language modules can be added and the algorithm used to generate alternate module handles without any central data housekeeping. As long as there is no collision between a new module name and an existing module name, the module and any code using it, or any binary file containing code and resources, may be added to the operating system without making any centralized changes.

The system automatically loads and frees alternate-language modules as necessary, and transparently to the user and the processes requesting resources. Alternate language resources reside in modules (dynamic link libraries or DLLs, as defined in Windows® parlance, in a preferred implementation), each uniquely specified by a path and module name as:

<module\_path>\mui\<language\_ID>\<module\_name>

In other words, the operating system loads an alternate-language resource module from a language-specific subdirectory of the original module's load path. The path and module name are dynamically generated using the same name as the original module name supplied by the calling process. The element <language\_ID> may be some compact code representing the language. For example, it could be based on ISO 639 language standard abbreviation plus, possibly, a sublanguage designator or a Win32 language id including primary and secondary components.

Alternate languages may be requested with varying degrees of specificity. That is, one may request (French) French, Swiss French, or Canadian French at

1 one level of specificity or just French at a lower level of specificity. For the  
2 process of generating an alternate language resource module handle to be robust,  
3 the algorithm may involve multiple steps to enable it to reconcile a system-level  
4 request for a user-interface language with one degree of specificity and an  
5 availability of alternate language resources provided with another degree of  
6 specificity. Suppose, for example, the user requests Swiss French upon logging  
7 into the operating system. This specifies a user-variable that mandates that for all  
8 process able to comply, that Swiss French resources should be used. The resource  
9 loader (or library loader) algorithm that generates alternate-language resources  
10 should be able to deal with situations where only an approximation to the  
11 requested language is available. Suppose in the above example, that only French  
12 and various other primary alternate languages are available and not specifically  
13 Swiss French. It is desirable for the algorithm to load the French alternate  
14 language resource upon a request rather than to make some other default choice  
15 that is not as close to the system-level mandate indicated in the system user  
16 language ID. Thus, multiple levels of approximation may be defined for the  
17 algorithm, for example, as follows.

18 First, the algorithm may determine if, in the module path specified by  
19 “<module\_path>\” there exists a subdirectory with an identifier equivalent to the  
20 current user language ID, that is, with the name “\mui\<language\_ID>\”. If this  
21 first test fails, the algorithm may determine if there exists a subdirectory of  
22 “<module\_path>\” with an identifier equivalent to the primary language ID  
23 corresponding to the current user language ID, that is, with the name  
24 “\mui\<primary\_language\_ID>\”. If no system user language ID is specified, the  
25 algorithm may be able to use a surrogate to resolve a subdirectory, for example,

1 some preference that suggests the locality of the user such as a preference as to  
2 date or monetary format conventions. Alternatively, a language-neutral alternate  
3 resource module may be invoked. Other steps, which may be placed in any  
4 desired priority, could be the selection of a default alternate language resource  
5 subdirectory, a substitute language where the one specified by the user language  
6 ID is not available but a fair substitute language spoken in the likely locale is. For  
7 example, if Canadian French is requested in the user language ID, and neither  
8 Canadian French nor French are available, but Canadian English is available, then  
9 the latter could be used. The above process of identifying preferred alternate  
10 resources according to a priority system allows the specificity of alternate  
11 language resources to be increased. If the operating system ships with only  
12 primary languages (e.g., English, but no British English, Canadian English, etc.)  
13 the user may add more specific languages later and the user's choice implemented  
14 transparently and automatically.

15 To speed processing, the mapping obtained by generating each alternate  
16 module path dynamically is preserved in a look-up table. When a calling process  
17 calls the same resource, the alternate resource module may be obtained from the  
18 look-up table instead of generating the path and handle dynamically. Note that by  
19 preserving the result of the dynamic generation of an alternate resource module  
20 ID, the steps of the robust algorithm discussed above do not have to be repeated  
21 each time a request for a resource is made.

22 In addition, a clean-up table is generated to help the modified resource  
23 loader load and free memory as system requirements permit. The clean up table  
24 lists the loaded alternate resource modules and the processes that requested them.  
25

1 When, for example, the process requesting a resource is terminated, the resource  
2 module requested by the terminated process may be unloaded from memory.

3 In an alternative embodiment, alternate-language resource modules are  
4 distinguished by different filenames within a single directory, rather than by  
5 placing resources modules of different languages in different directories. More  
6 specifically, different extensions are added to the filenames to indicate the  
7 languages of the resource modules.

8 Note that the operating system keeps track of resources that are loaded and  
9 unloaded by generating entries in a loader data table. The loader data table  
10 indicates the processes that required the loading of resource modules so that these  
11 modules can be unloaded when the process terminates or as other system  
12 requirements indicate. For modules that are loaded by the applications directly  
13 using, for example in Windows NT, the LoadLibraryEx function, the module's  
14 identity may not be "known" to the resource loader described above. That is, no  
15 loader data table entry is generated. In this case, the facility that loads the  
16 resource module (e.g., LoadLibrary) may inquire as to the existence of an  
17 alternate-language resource and load it instead of the module requested by the  
18 application. If the application or process does use an operating system facility that  
19 does generate a loader data table entry, then the module would not have to be  
20 loaded until a request is made for a resource from the resource loader by the  
21 application or other process.

22 According to an embodiment, the invention is a method performed by an  
23 operating system. The method redirects a call by a calling process for a first  
24 datum residing in a first binary file. The following steps are performed: storing in  
25 an operating user-setting independently of the calling process, a language

1 identifier; when a second binary file corresponding to the language identifier and  
2 also to an identifier of either the first datum or the first binary file exists: (1)  
3 dynamically generating a path to the second binary file responsive to the  
4 language identifier and the either the first datum or the first binary file; (2) storing  
5 the path in a look-up table correlating a process module identifier identifying the  
6 first binary file and an alternate module identifier identifying the second binary  
7 file; and (3) making an alternate datum in the second binary file accessible to the  
8 calling process instead of the first datum.

9 According to another embodiment, the invention is also a method  
10 performed by an operating system. The method redirects a call by a calling  
11 process for a first resource datum residing in a first binary file containing both  
12 executable code defining the calling process and resource data. The calling  
13 process is defined in the code. The method has the following steps: storing in a  
14 variable, independently of the calling process, a language identifier; when a  
15 second binary file corresponding to the language identifier and also to either the  
16 first resource datum or the first binary file exists: (1) dynamically generating a  
17 path to the second binary file responsive to the language identifier and the either  
18 the first resource datum or the first binary file; (2) making an alternate resource  
19 datum in the second binary file accessible to the calling process instead of the first  
20 resource datum.

21 According to still another embodiment, the invention is a method of adding  
22 multilingual capability to an operating system having functions to address first  
23 resource data in executable binary files. The method includes the following steps:  
24 adding a selectable user-setting for storing a selected language identifier; adding at  
25 least one alternate language resource file containing resource data each

1 corresponding to a respective one of the first resource data; and modifying a  
2 resource loader to redirect calls for each of the first resource data to a respective  
3 one of the alternate language resource data responsively to a selected language  
4 stored in the selected language identifier.

5 According to an embodiment, the invention is a method performed by an  
6 operating system. The method addresses data responsively to a call by a calling  
7 process for a first datum. The method has the following steps: determining an  
8 existence of an alternate language file corresponding to the first datum; returning  
9 at least one datum from the alternate language file to the calling process when a  
10 result of the step of determining is an indication that the alternate language file  
11 exists; returning the first datum to the calling process when a result of the step of  
12 determining is an indication that the alternate language file does not exist.

13 According to an embodiment, the invention is a method performed by an  
14 operating system. The method redirects a call by a calling process for a first  
15 datum residing in a first binary file. The following steps are performed: storing in  
16 an operating system variable independently of the calling process for each user, a  
17 language identifier; responsively to a detection of a second binary file  
18 corresponding to the language identifier and also to an identifier of either the first  
19 datum or the first binary file: (1) dynamically generating a path to the second  
20 binary file responsively to the language identifier and the either the first datum or  
21 the first binary file; (2) storing the path in a look-up table correlating a process  
22 module identifier identifying the first binary file and an alternate module identifier  
23 identifying the second binary file; and (3) making an alternate datum in the second  
24 binary file accessible to the calling process instead of the first datum.

1        Optionally, individual modules may contain shared-resource references.  
2        Each such reference is a pointer to a set or group of resource modules. Requests  
3        for resources from a module containing such a reference are redirected to the set of  
4        reference modules referenced by the shared-resource reference. Within the set, a  
5        particular module is selected based on the current language identifier maintained  
6        by the operating system. This allows a plurality of modules to use the same set of  
7        resource modules, and reduces the number of separate resource modules that need  
8        to be provided. More specifically, this allows all system resources to be specified  
9        in a single resource module, and to be shared by all other modules in the system.  
10      This reduces resource overhead and allows the operating system to load resources  
11      faster.

12      **BRIEF DESCRIPTION OF THE DRAWINGS**

13      Fig. 1 is a schematic drawing of a binary file containing a code section that  
14      defines a process that calls for a resource in a resource section of the same binary  
15      file.

16      Fig. 2 is a schematic drawing of two binary files, one of which contains  
17      code and may or may not contain resources, and the other of which contains  
18      resources and may or may not contain code, the code of the first file defining a  
19      process that calls resources in the second file.

20      Fig. 3 is a schematic illustration of a resource loader and resource finder  
21      being used by a process to retrieve a resource according one embodiment of the  
22      prior art.

23      Fig. 4 is a schematic illustration of a resource handler in a generalized  
24      description of a process of retrieving resources by a process on a computer.

1       Fig. 5 is a schematic illustration of a process of calling for a resource datum  
2 through an operating system in a modification of the prior art process shown in  
3 Fig. 3.

4       Fig. 6 is a block diagram showing an organization of modules in  
5 accordance with an embodiment of the invention.

6       Fig. 7 is a flowchart showing steps performed in accordance with an  
7 embodiment of the invention.

8

## 9 **DETAILED DESCRIPTION**

10     Referring to Fig. 5, a process of calling for a resource datum through an  
11 operating system in a modification of the prior art process shown in Fig. 3, is  
12 shown. Processes within the resource loader 130 and resource finder 135,  
13 described with reference to Fig. 3, are modified to produce a process as illustrated  
14 in Fig. 5. In overarching terms, the process of Fig. 5 redirects a call by the  
15 process for a particular resource to an alternate-language resource so that the  
16 process receives a resource associated with a selected user-interface language  
17 instead of the default resource for the process. In an embodiment, loading of  
18 alternate resources only “kicks in” if the process did not specify the language it  
19 wants to load. In other words, a process tries to load resources and doesn’t really  
20 care about which language. In the prior art system the resource loader would  
21 return the resources from either the resource section of the module itself, or from  
22 an external module, which the process specified to load resources from. In the  
23 present embodiment of a multilingual user interface system, the resource loader  
24 will load alternate resources if the process didn’t specify a particular language or  
25 other particular classification for the resource. The process requests a memory

1 handle from the resource finder 320 just as in the prior art embodiment of Fig. 3.  
2 However, in this case, the handle is a handle referring to an alternate language  
3 resource, if one is available. The resource finder tries to identify a resource  
4 indicated by a selected user-interface language ID 335.

5 Selected user-interface language ID 335 is a user-setting. The selected  
6 user-interface language ID 335 could be established, for example, by a user  
7 logging in and selecting a language from a list of options. The selected user-  
8 interface language ID 335 is then stored until changed.

9 A process 310 requests a memory handle for a resource by sending to a  
10 resource finder 320 a resource name and type. If the resource were in a module  
11 other than the one defining the calling process 310, the resource module handle  
12 would also be sent to the resource finder 320. If the module handle is not sent, the  
13 resource finder already has access to the module handle from a loader data table  
14 because the module is the same as the one generating the process requesting the  
15 resource. (As discussed in the background section, resource finder and resource  
16 loader are often used to access resources in the same binary file as the code  
17 generating the requesting process) It is also possible for the process to request a  
18 resource that is language-specific and the process of satisfying such a request may  
19 be outside the steps relating to the invention and satisfied by prior art methods  
20 (See for example a description of LoadResource described in  
21 <http://www.microsoft.com/msdn/>) In the latter case, a language ID may be passed  
22 to the resource finder.

23 The operating system is modified to maintain a table of alternate resource  
24 module handles 323, which have been generated previously by calls to the  
25 resource finder 320. So if another process already requested a resource from the

1 same module and the module is already correlated to an alternate resource module,  
2 the alternate module handle can be quickly obtained from the alternate resource  
3 module table 323. If there is no entry for the resource, the operating system  
4 generates an alternate module path dynamically.

5 To dynamically generate an alternate module path, an algorithm 325 is  
6 employed. The algorithm 325 may be based on some assumed organization of  
7 resource files, which indicate whether an alternate language resource file exists for  
8 the specified resource. In the present embodiment, the alternate language resource  
9 files are located in subdirectories of the requested module's path, each  
10 distinguished by a filename correlated uniquely to a language identifier. Within  
11 each language's subdirectory are stored the alternate language resource files, each  
12 named after the original module.

13 <module\_path>\mui\<language\_ID>\<module\_name>

14 In other words, the operating system loads an alternate-language resource  
15 module from a language-specific subdirectory of the original module's load path.  
16 If the original module, for a system that was not multilingual enabled, was "<path  
17 1>\<filename 1>," the path for the alternate language module would be "<path  
18 1>\mui\<language ID 1>\<filename 1>" assuming the language indicated by  
19 selected user-interface language ID 335 is "language ID 1."

20 The organization of the alternate language resources can be done in various  
21 alternative ways. Breaking them down into language-specific modules, each  
22 corresponding to the regular module (the one ordinarily requested in a single-  
23 language operating system) avoids any need for additional memory as would  
24 occur if for each resource module, the resources of the various languages were  
25 combined into a single module.

Given the path structure used to store modules, it is straightforward to construct a path for an alternate language module corresponding to any language indicated by the selected user-interface language ID 335 and the original called-for path and module name. This path is used by the resource finder 320 to provide a resource handle. The creation of the resource handle is done in the same way as in the prior art. The difference is that the resource handle in this instance directs the process to a resource datum 350, which was identified in a subdirectory of the original module path. In Fig. 5, the resource datum 350 was in an alternate resource module for “binary file 2” where the selected user interface language ID was language ID 2.

The path and module name are dynamically generated using the same name as the original module name supplied by the calling process. The element <language\_ID> may be some compact code representing the language. For example, it could be based on ISO 639 language standard abbreviation plus, possibly, a sublanguage designator or a Win32 language id including primary and secondary components.

In a preferred embodiment of the invention, the algorithm is robust in that it does much more than simply construct a path assuming there exists an alternate-language resource for the requested data. Alternate languages may be requested with varying degrees of specificity. Also, it is possible that no alternate-language resource may be available or that an alternate resource is available, the resource being different from the base resource in some respect other than language. The algorithm and associated processes are robust enough to deal with and exploit these situations as well as the straightforward scenario depicted in Fig. 5.

1       The selected user interface language may be very specific. For example, a  
2 user may request French, Swiss, or Canadian French. The algorithm may involve  
3 multiple steps to enable it to reconcile a system-level request for a user-interface  
4 language with one degree of specificity and an availability of alternate language  
5 resources provided with another degree of specificity. If the user requests French  
6 French upon logging into the operating system, only an approximation to the  
7 requested language may available. To deal with such situations, the algorithm and  
8 associated processes may operate according to a built-in hierarchy of steps as  
9 follows.

10      First, the algorithm may determine if, in the module path specified by  
11 “<module\_path>\mui\” there exists a subdirectory with an identifier equivalent to  
12 the current user language ID, that is, with the name “\<language\_ID>\”. If this  
13 first test fails, the algorithm may determine if there exists a subdirectory of  
14 “<module\_path>\muil\” with an identifier equivalent to the primary language ID  
15 corresponding to the current user language ID, that is, with the name  
16 “\<primary\_language\_ID>\”. If no system user language ID is specified, the  
17 algorithm may be able to use a surrogate to resolve a subdirectory, for example,  
18 some preference that suggests the locality of the user such as a preference as to  
19 date or monetary format conventions. Alternatively, a language-neutral alternate  
20 resource module may be invoked. Other steps, which may be placed in any  
21 desired priority, could be the selection of a default alternate language resource  
22 subdirectory, a substitute language where the one specified by the user language  
23 ID is not available but a predefined substitute language is often spoken in the  
24 likely locale is, for example, if Canadian French is requested in the user language  
25 ID, and English is available. The above process of identifying preferred alternate

resources according to a priority system allows the specificity of alternate language resources to be increased. If the operating system ships with only primary languages (e.g., English, but no British English, Canadian English, etc.) the user may add more specific languages later and the user's choice implemented transparently and automatically.

Note that the above functionality does not interfere with the normal requests for a resource for a specific language, such as made with the FindResourceEx function in Windows®. If a specified language ID is provided by the requesting process, the alternate-language resource scheme above would not reroute the request to another resource module.

After the algorithm 325 that forms the path has settled on a resource path, version checks and any other integrity checks can be performed on the identified file before making it accessible to the requesting process. If, as a result of the processes described with reference to Fig. 5, the alternate-language module 370 was newly placed in memory or otherwise made accessible by the call to resource finder 320, a new entry may be placed in the alternate resource module table 323. Finally a handle may be returned to the calling process to allow the process to access the requested resource. The latter may involve a step to another function, the resource loader 330, to load the data into memory and provide a handle for the process to use to access the data.

Note that where Fig. 5 and the attendant discussion indicate that the module is loaded into memory, this may not need to be done explicitly by the resource finder or even the resource loader. The only requirement is the appropriate data is made available to the process. The operating system may handle the actual movement of data through its I/O and memory management facilities. The import

of what is described above with reference to Fig. 5 is that a request by a process for a resource, whose guts are different for different languages, is automatically redirected transparently to the requesting process. The code defining the process does not need to be modified for the operating system to be multilingual enabled. Fig. 5 and the attendant discussion describe the process of redirecting requests for data in the context of resources that are incorporated in binary files that also contain executable code. The same basic formula can be expanded to embrace the access of data in resource-only files, for example, DLLs.

Note that in the above discussion, where a process calls for data to be loaded into memory or unloaded from memory, such a step should be viewed in the broader sense of being mapped into the address space of a process. This is because the operating system facilities for mapped I/O blur the concrete notions connected with loading data from disk into memory. In other words, current operating systems make it possible to allow a process to access data on a disk following steps without necessarily being involved in the explicit steps of loading data into memory, since this concrete step can be handled transparently by the operating system's I/O system and virtual memory management functions.

The above process may map the alternate resource module as a simple data file into the address space of the calling process. The details underlying this process are known in the prior art, for example in Windows®, this is done by code defining an operating system function called LoadLibrary.

As mentioned above, there are various ways of differentiating between resource files corresponding to different languages. For example, instead of manipulating file system paths it might be desirable in some situations to manipulate file names themselves. In accordance with one embodiment of the

1 invention, resource files consist of two parts: an initial part which is identical to an  
2 executable module with which the resource module corresponds, and a language  
3 extension which indicates the language of the localized resources within the  
4 resource module.

5 Fig. 6 illustrates how resource modules are associated with corresponding  
6 executable modules in accordance with this embodiment. As background, many  
7 modules are written using either the “C” or “C++” programming languages. In  
8 accordance with accepted practice, resources are commonly defined in separate  
9 source files, referred to as “resource script files,” having an “.rc” extension rather  
10 than the “.c” extension that is used for other executable modules. Resource script  
11 files are compiled with a resource compiler (normally named “RC.EXE”) which  
12 produces a “.res” file (a file having a “.res” extension).

13 Resource modules are typically paired with normal executable modules; the  
14 executable module contains executable code, while the resource module contains  
15 resources used by the executable code. At link time, the resource file is bound to  
16 the end of the executable file, and all of the resources defined in the resource script  
17 file become part of the executable file, for use during execution. As described  
18 above, the “FindResource” and “LoadResource” functions are used by an  
19 executable module to load resources either from the executable module itself or  
20 from another identified executable module. When invoking the “FindResource”  
21 function, the calling module identifies the executable module in which the desired  
22 resources are defined.

23 In accordance with one embodiment of the invention, a computer system  
24 comprises a plurality of executable modules 401, 402, 403, and 404. For purposes  
25 of discussion, these executable modules are named “myapp1.dll”, “myapp2.dll”,

1 “myapp3.dll”, and “myapp4.dll”. In addition, the computer system comprises a  
2 plurality of resource modules 411, 412, 413, 414, 415, and 416, corresponding to  
3 multiple languages. Each resource module contains localized resources for the  
4 corresponding language. Note that these resource modules, in practice, are normal  
5 executable modules except that they contain no executable code—just defined  
6 resources. Note also that executable modules 401-404 might contain resource  
7 definitions (although they will not normally be used in this embodiment of the  
8 invention).

9 The resource modules 411-416 are named to form sets 417, 418, and 419.  
10 All of the resource modules in a given set define the same actual resources, but  
11 each module in a set defines its resources in a different language. All of the  
12 resource modules in a given set are named in two parts. The first part of the  
13 filename is a conventional “.dll” filename. This part of the overall filename is the  
14 same for all of the modules within a set, indicating that the modules are part of a  
15 set. The last part of the filename is an extension comprising a language code  
16 followed by a “.mui” extension. This part of the filename indicates the language  
17 of the included resources. In Fig. 6, the language code comprises a descriptive  
18 textual designation of the language. In actual embodiment, the language code is a  
19 numerical code that indicates a particular language.

20 In accordance with the invention, a resource request to the operating  
21 system’s resource manager will identify a particular module such as module 401,  
22 402, 403, or 404. Rather than using the resources defined in that module,  
23 however, the resource manager will identify a set of resource modules that  
24 corresponds to the module identified in the resource request, and provide resources  
25 from one of the resource modules of that set.

1       Fig. 6 illustrates how a correspondence is set up between an executable  
2 module and a set of resource modules. This can be done in two ways. The first  
3 type of correspondence is illustrated by modules 401 and 402 and corresponding  
4 resource module sets 417 and 418. When using this type of correspondence, each  
5 executable module has a corresponding unique set of resource modules. In this  
6 case, executable module 401, named “myapp1.dll”, has a corresponding set 417 of  
7 resource modules which in turn contains resource modules 411 and 412, named  
8 “myapp1.dll.french.mui” and “myapp1.dll.english.mui”. Note that the first part of  
9 the resource module filenames is identical to the filename of the corresponding  
10 executable module. This is what identifies resource modules 411 and 412 as  
11 belonging to a set corresponding to executable module 401.

12      Similarly, executable module 402, named “myapp2.dll”, has a  
13 corresponding set 418 of resource modules which in turn contains resource  
14 modules 413 and 414, named “myapp2.dll.french.mui” and  
15 “myapp2.dll.english.mui”.

16      When an application program or executable module requests a resource  
17 from the resource manager, and identifies a particular executable module such as  
18 module 401 from which the resource is to be obtained, the operating system  
19 resource handler first identifies a set of resource modules associated with the  
20 identified executable module. This is done by noting the filename of the identified  
21 executable module, such as “myapp1.dll”. Any resource module whose filename  
22 begin with this string belongs to the set of resource modules associated with the  
23 identified executable modules. In this case, resource modules 411 and 412 have  
24 filenames beginning with “myapp1.dll”, and thus belong to the desired set of  
25 resource modules, corresponding to identified module 401.

1       Next, the resource handler selects one of the sets of resource modules in  
2 accordance with a language identifier that is obtained independently of the request  
3 to the resource handler. This is done by constructing a resource module filename  
4 that begins with the filename of the identified module (“myapp1.dll”) and then  
5 continues with an extension formed by the language code corresponding to the  
6 current language identifier, followed by “.mui”. Thus, if the current language  
7 identifier and corresponding language code is “french”, the resource handler  
8 constructs a filename “myapp1.dll.french.mui”. The resource manager then  
9 retrieves the requested resource from this file and provides it to the requesting  
10 application program or executable module.

11      A second type of correspondence between executable modules and resource  
12 modules is illustrated by executable modules 403 and 404, and resource module  
13 set 419. In accordance with the invention, some executable modules contain  
14 shared-resource reference. Each shared resource reference identifies a set of  
15 resource modules corresponding to different languages. In the described  
16 embodiment, the shared-resource reference indicates a portion of a filename or file  
17 identifier. In the example shown in Fig. 6, executable module 403 and 404 each  
18 contain a shared resource reference 420. The appropriate set of resource modules  
19 is identified by a string that specifies the first part of a resource file’s filename,  
20 such as “shared.dll” in the example. In this case, the different executable modules  
21 403 and 404 contain the same shared-resource reference: “shared.dll”. Thus, each  
22 of these two executable modules points to a single, shared set of resource modules  
23 419.

24      When an application program or executable module requests a resource  
25 from the resource manager and identifies a module such as executable module 403

1 or 404 from which the resource is to be obtained, the resource handler responds by  
2 identifying a set of associated resource modules. This is done by selecting or  
3 constructing a filename in accordance with both the shared-resource reference and  
4 the current language identifier maintained by the operating system. In particular, a  
5 filename is constructed by appending the shared-resource reference (in this case  
6 “shared.dll”) with the extension “.language.mui”, where “language” is a language  
7 code corresponding to the current language identifier. The resource handler then  
8 provides the requested resource from the resource file having the constructed  
9 filename.

10 This scheme allows resource modules to be consolidated in a project having  
11 multiple executable modules. Rather than having a different English-language  
12 resource module for every single executable module, a single English-language  
13 resource module can support a plurality of executable modules. This greatly  
14 reduces the number of separate resource modules required in a complex  
15 application.

16 Fig. 7 illustrates steps performed in this embodiment of the invention. A  
17 step 430 comprises receiving a request for a resource. This request is typically  
18 made by an application or executable module to the resource handler, through the  
19 “FindResource” and “LoadResource” functions discussed above. As part of the  
20 request, the application specifies the name or ID of the requested resource and also  
21 provides a module identifier indicating the module from which the resource is to  
22 be obtained. The module identifier is an operating system handle that corresponds  
23 to the desired module. The desired module has a filename known to the operating  
24 system.

1       A step 431 comprises determining whether the module identified by the  
2 module identifier includes a shared resource reference. In the described  
3 embodiment, a shared-resource reference is implemented as a resource with a  
4 predefined or reserved name. The described embodiment uses the resource name  
5 “RT\_MUI”. The value of this resource is set to a character or Unicode string that  
6 specifies the first part of a resource module filename. For example, the  
7 “RT\_MUI” resource might be set to “shared.dll” as in Fig. 6.

8       If step 431 finds the existence of the “RT\_MUI” resource within the  
9 module identified in the resource request, a step 432 is performed of selecting a  
10 resource module of the appropriate language from a set of resource modules  
11 identified by the “RE\_MUI” resource value. In the described embodiment, the  
12 “RT\_MUI” value comprises the first part of a filename, and thus identifies a set of  
13 files that all start with that same value. The module of the appropriate language is  
14 selected by appending an extension to the “RT\_MUI” shared-resource reference.  
15 The extension consists of the string “.language.mui”, where “language” is a  
16 language code that corresponds to the language indicated by the operating  
17 system’s language identifier.

18       If step 431 does not find a shared-resource indicator in the module  
19 identified with the resource request, a step 433 is performed of selecting a resource  
20 module of the appropriate language from a set of resource modules identified by  
21 the filename of the identified module. The identified module’s filename is used as  
22 the first part of a resource module filename, and thus identifies a set of files that  
23 all start with that same string. A particular module is selected by appending an  
24 extension to the identified module’s filename, consisting of the string

1 “.language.mui”, where “language” is a language code that corresponds to the  
2 language indicated by the operating system’s language identifier.

3 Step 434 follows step 432 or 433, and comprises providing the requested  
4 localized resource from the resource module identified by the constructed  
5 filename.

6 In this manner, resource of the appropriate language are provided, without  
7 any overt actions by the requesting module to select a particular language.

8 In some cases, the language indicated by the current language identifier  
9 might not be represented by the resource modules of a particular set of resource  
10 modules. To accommodate this, the resource handler provides a fallback  
11 mechanism that attempts to provide the best possible choice of languages, even  
12 though the exact language indicated by the language identifier is not available.  
13 Specifically, if none of the resource modules accords with the language identifier,  
14 the resource handler selects an alternative resource module in accordance with the  
15 following languages, in descending order of priority:

16 the primary language of the language indicated by the language  
17 identifier;

18 the system’s default language;

19 the primary language of the system’s default language;

20 US English;

21 primary English.

22 The term “primary” refers to a general language which includes a plurality  
23 of subsets. For instance, a system might utilize “Canadian French.” The primary  
24 language of Canadian French is simply French. A “system default” language is set  
25

1 when a computer is initially configured, and might vary from the language  
2 currently indicated by the operating system's language identifier.

3 Although the invention has been described in language specific to structural  
4 features and/or methodological steps, it is to be understood that the invention  
5 defined in the appended claims is not necessarily limited to the specific features or  
6 steps described. Rather, the specific features and steps are disclosed as preferred  
7 forms of implementing the claimed invention.

8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25